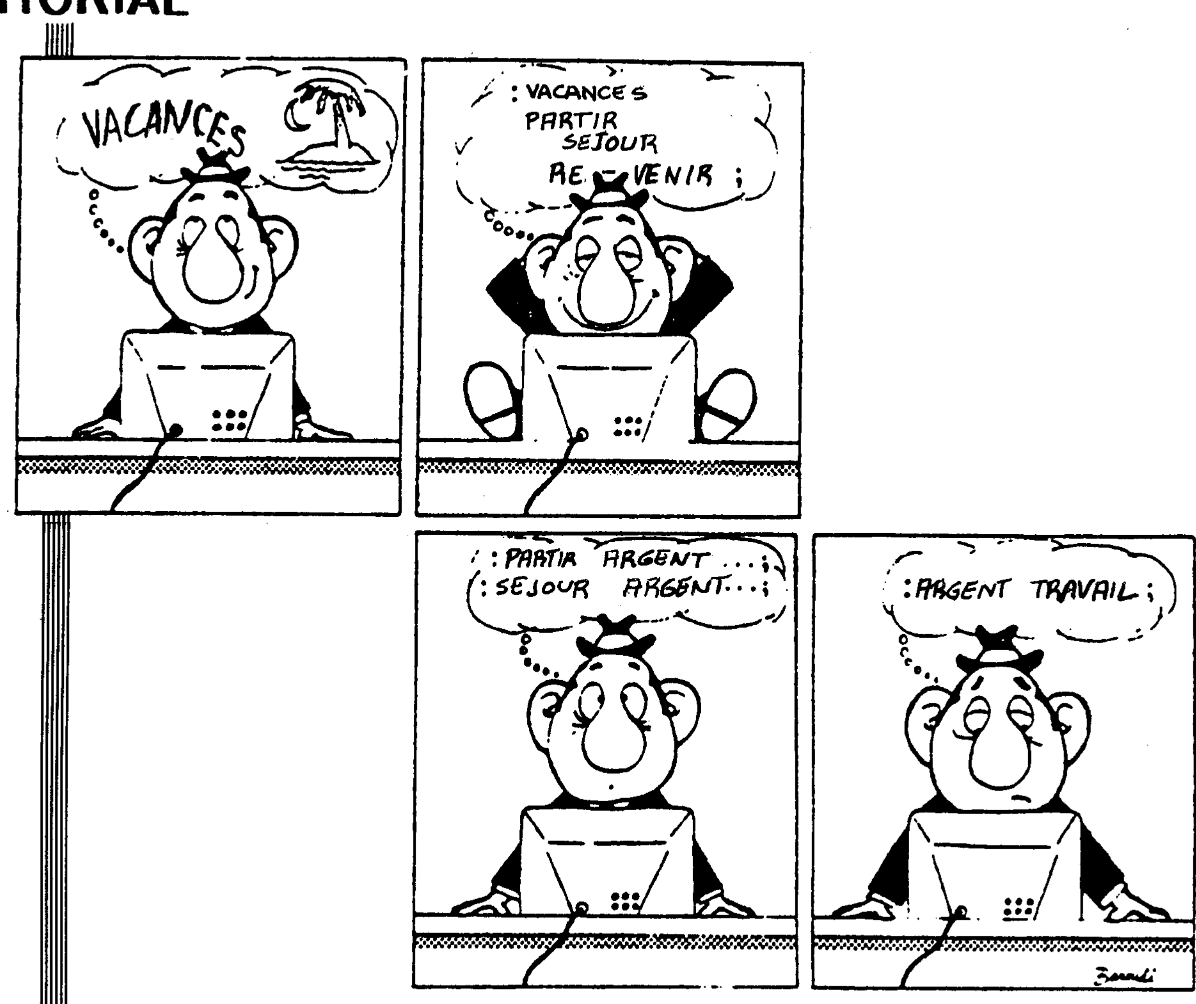


26

MUMPS, FORTH, LPB, FUTURLOG...

JUILLET 1986

EDITORIAL



SOMMAIRE

ASSOCIATION JEDI 8, rue Poirier de Narçay 75014 PARIS
Tel: (1) 45.42.88.90 (de 10h à 18h)

XIV LES ENTREES/SORTIES

A) Introduction

Nous avons déja vu les deux uniques commandes d'entrées/sorties dispon ibles en MUMPS. Lorsque nous les avons utilisées, elles permettaient de dialoguer avec le terminal de l'utilisateur. Mais, MUMPS permet d'effectuer des entrées/sorties (I/O ou input/output) sur d'autres périphériques tel que des imprimantes, des dérouleurs de bandes et même d'autres ordinateurs. Pour réaliser ces communications,les ordres READ et WRITE seront, de la même manière, utilisées avec une syntaxe étendue.

B) La commande OPEN

Avant d'utiliser un péripherique, il est indispensable, dans un contexe multi-utilisateurs, d'avertir la machine que celui-ci est réservé pour un programme. Cette demande de réservation est effectuée à l'aide de la commande OPEN. Bien entendu cette demande est honnorée si le péripherique demandé est dispon ible. La commande OPEN est accompagnée de trois arguments dont la signification est la suivante :

- Une expression (ou un nombre entier) correspondant au péripherique ou au fichier à réserver.

- Une ou plusieurs expressions, appelées "paramètres du periphérique" qui fournissent les informations nécessaires à la description du periphérique ou du fichier.

- Une expression qui indique le temps maximum (nombre de secondes) pendant lequel on accepte d'attendre la dispon ibilite du périphérique.

Puisque les périphériques peuvent changer d'une machine à l'autre, le standard MUMPS ne précise pas le contenu de la partie "paramètres du périphérique". L'utilisateur sera obligé de se reporter à la documentation de son installation. Mais MUMPS est capable de dire au programme exécutant un OPEN si la demande est satisfaite ou non. Ceci a l'aide de la variable système $TEST que nous avons déjà étudié . Cette variable est alimentée avec la valeur 0 (faux) si après le temps maximum inparti, le périphérique n'est pas ouvert. Dans le cas contraire $TEST contiendra 1 (vrai). La variable $TEST est affectée, seulement, si l'on utilise le paramètre de temps.

Afin d'illustrer cette commande admettons que notre configuration comporte un terminal-écran, une imprimante et un lecteur de carte. Les numéros attribués a ces périphériques étant les suivants :

Le numéro 0 pour l'écran
Le "" 1 "" l'imprimante
Le "" 2 "" le lecteur de carte

Note : Sur la plupart des implantations, il n'est pas nécessaire d'ouvrir le terminal de l'utilisateur.

Imaginons, que votre programme a besoin d'utiliser l'imprimante et le lecteur de carte. L'une des deux commandes suivante est nécessaire :

a- OPEN 1:pi,2:pl ou O 1:pi,2:pl ou O 1:pi O 2:pl

b- OPEN 1:pi,2:pl:60

Les indications pi et pl représentent les paramètres de l'imprimante et du lecteur de carte. Dans la deuxième commande, on peut noter l'apparition du paramètre 60. Il indique a MUMPS d'alimenter la variable $TEST et de compter le temps. Si au bout de 60 secondes le lecteur de carte n'a pas pu être ouvert $TEST contiendra 0 (faux). A l'inverse, si l'OPEN a pu s'effectuer, $TEST contiendra 1 (vrai).

C) La commande USE

Le fait d'avoir ouvert un périphérique, n'indique pas qu'il est utilisé, mais simplement qu'on se le réserve. Maintenant, nous allons étudier un nouveau concept. Il concerne ce qu'on appele le périphérique "courant". Les ordres READ et WRITE font toujours appel au dernier périphérique référencé. Initialement, le périphérique courant est la console de l'utilisateur. Lorsqu'un programme désire changer de prériphérique courant, il utilisera la commande USE (U). L'argument de la commande USE est le numéro de périphérique. Il est possible, comme dans la commande OPEN, de spécifier des paramètres du périphérique. Bien entendu, il serait stupide de mettre plusieurs numéros de phériphérique dans la commande USE puisque celle-ci definit LE périphérique COURANT. Imaginons que l'on veuille écrire sur l'imprimante, nous pouvons écrire la routine suivante (en conservant les numéros de périphérique définis au paragraphe précédent) :

```
DEB       ;ESSAI IMPRESSION   Y.L.G.  2/OCT/84  3/OCT/84
          W !,"écriture,à la ligne, de ce texte"
          O 1 ;ouverture de l'imprimante
          U 1 ;utilisation de l'imprimante comme périph. courant
          F I=1:1:10 W !,"Numéro de ligne : ",I
          Q
```

D) La commande CLOSE

Afin de ne pas monopoliser un périphérique "éternellement" un nouvel ordre permet d'avertir la machine qu'il redevient dispo nible. Cette commande est l'ordre CLOSE (C). La syntaxe et les paramètres sont les mêmes que pour les ordres OPEN et USE. Apres l'exécution d'un ordre CLOSE, le péripherique courant redevient, généralement, la console de l'utilisateur.

E) Les variables systèmes $X, $Y, $IO

Comme nous l'avons déja vu, le système est capable de maintenir deux variables alimentées par les ordres d'écriture ($X pour les colonnes, $Y pour les lignes). Ces variables sont, toujours, relatives et uniques au périphérique ou au fichier courant. Souvenez-vous des signes cabalistiques qui permettent de gérer les formats d'écriture (!, ?, #). D'autre part, la variable systeme $IO (ou $I) est alimentée par la commande USE. Elle contient le numéro du périphérique courant.

F) Extensions relatives aux commandes READ et WRITE

Ce paragraphe nous permet d'introduire des arguments nouveaux aux commandes READ et WRITE, donc de nouvelles fonctionnalités. La première de celles-ci, permet d'entrer ou d'écrire un et un seul caractère, en prenant en considération sa valeur décimale de la table ASCII. Pour ce faire, la variable réceptrice (READ) ou emmettrice (WRITE) sera précédée d'une étoile (* ou asterisque). Dans le cas du READ, c'est la méthode à employer pour saisir au clavier un caractère de contrôle. Ainsi, si on écrit l'ordre suivant et que l'entrée au clavier est la touche <tab>, ou ctrl-I, le contenu de la variable réceptrice (VAR) est égal à 9.

```
READ *VAR
```

Nous pouvons reprendre le même exemple avec la commande WRITE. Cette fois-ci, nous pouvons donnerune équivalence afin de mieux comprendre.

```
WRITE *9   <===== ou =====> WRITE $C(9)
```

Le nouveau standard (de 1982) a inclus une autre extension à la commande READ. Il est souvent nécessaire de limiter la longueur de saisie au clavier à un certain nombre de caractères. Pour ce faire, un nouvel argument est à ajouter au READ. Il s'agit du signe # suivi du nombre de caractère maximum. Donc en exécutant l'ordre READ CODEPOST#5, l'utilisateur ne pourra pas entrer plus de cinq caractères au clavier lors de l'execution.

Ayant vu dans les précédents paragraphes l'utilisation du péripherique "courant", souvenez vous que les commandes READ et WRITE s'appliquent à celui-ci. Dans ces conditions, il est indispensble de pouvoir fixer un temps maximum d'attente d'entrée. Aussi ce temps peut être défini dans la commande READ. Pour ce faire, il suffit de préciser en dernier argument le nombre de secondes pendant lequel on accepte d'attendre. Imaginons que l'on veuille une saisie d'un mot de passe de trois caractères au clavier et que le temps maximum autorisé pour la reponse soit de dix secondes, nous écrirons :

```
READ !,"entrez votre mot de passe : ",MOT#3:10
```

La variable $TEST est affectée aussitôt que l'on utilise un temps limite. Dans le cas ou l'opérateur n'a pas repondu dans le temps prévu, la variable $TEST ($T) contient 0 (faux) autrement elle contient 1 (vrai). Une valeur particulière affectée à $TEST est prévue pour la commande READ d'un seul caractère avec un temps limite (READ *variable:temps). Sur certaines implémentations, si le temps est dépassé, c'est la valeur -1 qui est affectée à $TEST.

Note : La commande READ n'autorise la saisie que des caractères visualisables. Si la saisie doit incorporer des caractères de contrôle, la forme READ *variable doit etre utilisée.

G) La fonction $JUSTIFY

Nous avons vu que l'utilisation du signe ? suivi d'un entier dans un ordre WRITE, permettait de se positionner une colonne specifiée. Ceci permet de cadrer le texte à gauche . Cette fonctionnalité est appelée : "justification à gauche". De temps en temps, il est intéressant d'aligner sur la droite du texte ou des valeurs. On dit qu'il s'agit d'une "justification à droite". Admettons que l'on veuille cadrer à droite à la colonne 20 le texte "bonjour". Nous serions obligés d'écrire les lignes suivantes :

```
F I=0:1:20-$L("bonjour") W " "
W "bonjour"
```

Cette forme d'écriture nécessite beaucoup de place et un long temps d'exécution. Aussi MUMPS met à notre disposition une fonction qui permet la justification à droite, c'est la fonction $JUSTIFY ($J). En utilisant le même exemple, nous obtenons le même résultat en écrivant :

```
W $J("bonjour",20)
```

Cette fonction a d'autres domaines d'utilisation. On peut, grâce à elle, créer un littéral de toute pièce. Le fait d'écrire S DIXBLANC=$J("",10) permet d'affecter à la variable DIXBLANC 10 espaces. Nous pouvons voir, de part la structure de $JUSTIFY, que le deuxième argument correspond à la zone maximum de cadrage a droite.

Attention ! Si la longueur de la chaîne dépasse la valeur du 2ème argument, le cadrage ne sera pas effectué.

Un troisième argument peut être ajouté dans la fonction $JUSTIFY. Il sert à "justifier" et/ou "arrondir" une valeur numérique. Pour le moment cet argument doit TOUJOURS être positif. Par la suite les concepteurs de MUMPS utiliseront peut être les valeurs négatives pour des actions particulières. Les exemples suivants sont suffisament explicites pour ne pas faire de longs discours.

expression a evaluer	valeur produite
$J(3.1416,0,0)	3
$J(6.512345,0,0)	7
$J(4.325,0,2)	4.33
$J(5.76,0,1)	5.8
$J(2.482,4,2)	2.48
$J(2.482,0,5)	2.48200
$J(7,0,2)	7.00
$J("ABC",2,0)	0
$J("XYZ",0,0)	0
$J("GGG",5,3)	0.000
$J(.5631,4,2)	0.57

Dans le cas d'une valeur fractionnaire un zero non significatif est ajoute. Cette caracteristique est tres interessante dans les formats de sortie requis en gestion.

H) La commande JOB

Une des utilisations les plus frequentes des ordinateurs, bien que souvent pas necessaire, est de sortir du papier. Pour ce faire, les programmes n'ont pas besoin d'etre interactifs, il n'est donc pas necessaire de bloquer la console de l'utilisateur pendant une tache d'edition. Puisque MUMPS est par essence un systeme multi-taches, il est naturel de pouvoir lui sous-traiter une tache "fantome" (ou tache auxilliaire). Plus generalement, on emploi une tache "fantome" ("background") lorsqu'il n'y a pas de dialogue avec l'operateur. Cette sous-traitance est realise a l'aide de la commande JOB. MUMPS permet d'initialiser une ou plusieurs taches auxilliares qui s'executeront dans des partitions differentes de la machine. Si nous ecrivons, puis nous sauvegardons la routine suivante :

```
ESSAI      ;PRG. POUR JOB/Y.L.G. 20/11/84 20/11/84
DEBUT      O IMP:0 Q:'$T  U IMP ;IMP contient le numero de l'imprimante
           F I=1:1:60 W !,"boucle en cours  numero courant : ",I
           C IMP Q ;fin de la routine
```

Maintenant, nous voulons qu'elle s'execute dans une autre partition. Pour ce faire, il suffit d'ecrire la commande ci-dessous :

```
JOB ^ESSAI     ou     JOB DEBUT^ESSAI
```

Chaque fois qu'une commande JOB est specifiée, une nouvelle partition est requise. Le périphérique courant de la tâche background est, sauf indication contraire, la console de l'utilisateur qui a demandé cette exécution.

Des paramètres spécifiques peuvent être ajoutés à la commande JOB (Ex. la taille mémoire nécessaire pour l'exécution du programme). Ceux-ci seront séparés du reste de la commande par le signe : (deux points). Il est egalement possible d'ajouter un temps limite de prise en compte de la commande JOB. Dans ce cas, la variable $TEST est alimentée a: zero 0 (faux) si l'exécution de JOB n'est pas realisée dans le temps imparti, un 1 (vrai) dans le cas contraire. Exemple :

```
JOB ^ESSAI:20:1
```

On demande d'exécuter la routine ESSAI en background en lui réservant 20 Ko et en acceptant d'attendre 1 seconde.

RESUME

Dans ce chapitre, nous avons vu les extensions des commandes READ et WRITE. Comment un utilisateur pouvait accéder à un péripherique autre que son terminal à l'aide des commandes OPEN, USE et CLOSE. Ce qui permet d'affirmer que les télécommunications, entre les machines equipées de MUMPS, sont très simples à mettre en oeuvre. Ces types de communications pourront être implantées comme tâches "background" à l'aide de la commande JOB. Cette commande étant utilisée à chaque fois qu'un traitement ne demande pas l'intervention d'un opérateur.

 par Pierre CHATELAIN

Le système FUTURSYS a été présenté dans le numéro 19 du mensuel JEDI et par les revues MICRO-SYSTEMES et L'ORDINATEUR INDIVIDUEL. Mais pour JEDI, il ne suffit pas de présenter un système, il faut aussi parler des concepts mis en oeuvre par ce nouveau système dont la principale caractéristique est d'être pour le moment le seul système d'intelligence artificielle portable. C'est pourquoi, nous avons pris contact avec son créateur. Grâce à lui, nous sommes en mesure de vous proposer cette série.

LES FAITS

Les faits sont les connaissances élémentaires qui, à l'intérieur d'un concept, permettent d'évaluer une expression préalablement analysée, c'est à dire transformée en arbre.

Le quadruplet suivant définit le fait:
- une relation binaire, dite relation d'inférence.
- une expression.
- un développement de cette expression.
- une fonction implicite, éventuellement nulle, dont l'exécution est liée à l'utilisation du fait.

Les expressions utilisées étant des arbres, elles peuvent donc représenter des types quelconques, numérique, logique, formel, etc... (la logique pouvant être quelquonque, car définie par exemple au moyen d'autre faits).

L'utilisation d'une base de faits pour évaluer une expression consiste à comparer cette dernière, ou une partie de cette dernière, au second membre de chaque quadruplet-fait (unification), à la condition préalable que, pour une même expression, il puisse y avoir plusieurs faits correspondants à des relations d'inférence différentes. Exemple:

on pourrait avoir 1+2=3
(avec la relation .=.)
type (2):N
(avec la relation type(.):.)
1R3
(avec la relation .R.)

Si l'essai d'unification aboutit à un succès, alors l'expression ou sa partie entière est remplacée par le troisième membre du quadruplet-fait (c'est l'inférence), la fonction implicite dernier élément du quadruplet étant également activée.

Selon que l'on se satisfait d'une seule unification (un seul fait choisi) ou qu'au contraire on recherche toutes les unifications possibles, FUTURSYS fonctionne respetivement en mode algorithmique ou en mode combinatoire. Nous verrons plus loin l'intérêt de chacun de ces modes.

Mais au préalable, nous allons expliciter la représentation d'un fait qui s'établit comme suit:

NoI,plicite, expression en-relation-avec développement de l'expression.

La relation pouvant être par exemple:

```
Ai=Ai   Ai=>Ai   Ai<=Ai   Ai:Ai
type(Ai):Ai       F(Ai)=Ai
AlorsAiSiAi       AiImpliqueAi
                              etc...
```

C'est une relation binaire quelconque, définie en structure. Cette relation étant sélectionnée comme un paramètre de l'unification, il faut remarquer qu'elle permet donc de choisir certains faits seulement du concept (ceux comportant la bonne relation d'inférence). Le numéro de la fonction implicite nulle est 0.

Quelques exemples de faits:

```
 0, 1+2=3
13, A1+A2=A9
 0, SiVraiAlorsA1SinonA2=A1
 0, VraiOuA1=Vrai
 0, SiFauxAlorsA1SinonA2=A2
 0, VraiEtA1=A1
 0, FauxOuA1=A1
 0, FauxEtA1=Faux
 0, A1GP.A2<=A1PE.A3EtA3PE.A2OuA3ME.A2
 0, 0!=1
 0, A1!=(A1-1)!XA1
 0, PGCD(A1,A2)=SiReste(A2,A1)Vaut0AlorsA1Si
nonPGCD(Reste(A2,A1),A1)
 0, TourHanoi(A1,A2,A3)=SiA1Vaut0AlorsRienSi
nonTourHanoi(A1-1,A2,(6-A2)-A3);Ecrire"Disque
"uA1u"de"uA2u"à"uA3;TourHanoi(A1-1,(6-A2)-A3,
A3)
```

(déplacer une tour de hauteur A1 de la colonne A2 à la colonne A3).

FONCTIONS IMPLICITES

L'objet des fonctions implicites, qui sont activées lors de l'inférence, est d'effectuer soit des calculs lourds et formels (opérateurs numériques), soit des opérations d'entrée-sortie qui ne se prêtent pas au calcul formel. A cette fin, FUTURSYS propose à l'utilisateur une biliothèque de fonctions implicites permettant à celui-ci d'effectuer les entrées-sorties, le traitement des nombres, des chaînes, des arbres, ainsi que des fonctions du système.

UNIFICATION

Cette opération consiste à comparer une partie de l'expression à évaluer, au second membre d'un quadruplet-fait, selon un des deux critères suivants:

- l'identité.
- l'inégalité, ou différence.

Ces deux critères, mutuellements exclusifs, permettent de valoriser des arguments vides.

Ces mêmes arguments peuvent, et c'est en général leur intérêt, être utilisés par le troisième membre du quadruplet (développement de l'expression), ainsi que par le quatrième et dernier (fonction implicite). Avec la notation Ai représentant l'argument no i, on a les exemples suivants:

1+2 s'unifie à 1+2, 1+A3, A4+A5, mais pas à A1+A1, si on utilise le symbole #, qui permet de signaler une unification par différence, on a :

1+2 s'unifie à #Vrai, #0, mais pas à #X.

L'INFERENCE

L'inférence est une opération qui remplace l'expression unifiée, par le troisième membre du quadruplet-fait (le développement de l'expression), et qui active également la fonction implicite liée à l'utilisation du fait.

Dans le cas où il y a plusieurs faits unifiés (mode combinatoire), les inférences auront lieu tour à tour grâce à un mécanisme d'exploration de l'arbre des possibilités (d'unification), appelé 'back-tracking'. A la suite de l'inférence, l'exprsession résultat, ou une partie de celle-ci, ou encore l'expression résultat englobée dans une partie de l'expression initiale, subira le même traitement d'évaluation.

L'ALGORITHMIQUE

L'algorithmique permet de ne se satisfaire que d'une seule unification, même si plusieurs sont possibles à l'intérieur de la

Suite page 10

COMMENT SITUER LA PILE DE DONNEES EN MEMOIRE

La pile de données, en langage FORTH, est sans nul doute l'endroit le plus commode pour passer des données et valeurs d'une définition à une autre, que celles-ci soient définies en FORTH ou en langage machine. Mais il devient malaisé de gérer plus de trois ou quatre paramètres simultanément. Bien sûr, le mot PICK permet un accès à n'importe quelle donnée, surtout celles que les mots DUP, ROT, SWAP et DROP ne peuvent atteindre. Mais réfléchissons un peu. Voyons la situation où x données ont été empilées:

- une première manoeuvre à l'aide de n PICK, avec n compris entre 1 et x (et n non compris dans les x éléments), dépose au sommet de la pile la nième valeur contenue dans la pile de données.
- pour déposer la même valeur au sommet de la pile que précédemment, il faut maintenant taper n+1 PICK.
- imaginez ce que donne le programme quand on gère 5, 10, voire 15 paramètres ou plus dans la même définition. La lisibilité de la définition devient douteuse, pour vous comme pour les autres, et la gestion de tant de paramètres devient un cauchemar.

Or, le contenu de la pile de données peut être situé en mémoire à l'aide de deux adresses. La première, est accessible par S0 qui délivre l'adresse du début de la pile. La seconde par SP@ qui délivre l'adresse du pointeur de pile. Lorsque la pile est vide, les adresses délivrées par S0 et SPà sont identiques.

Sur la majorité des systèmes, la pile est implantée physiquement à partir du sommet de la mémoire et l'empilement des données décrémente la valeur du pointeur de pile.

Voici le diagramme d'une pile vide:

```
———— <- S0= adr0  <- SP@ = adr0
```

NOTA: si les contenus de S0 et SP@ affichent des valeurs négatives, utilisez le mot 'U.' pour afficher des données exprimant des adresses mémoires.

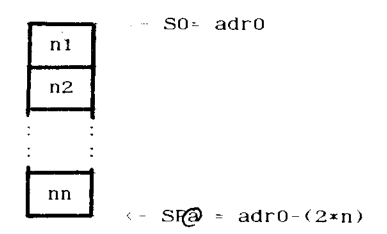
L'empilement d'une valeur 16 bits quelconque donne le résultat suivant:

```
 ┌────┐  S0= adr0
 │ n1 │
 └────┘  SP@  adr0-2
```

L'empilement d'une seconde valeur 16 bits quelconque donne maintenant:

```
 ┌────┐ <- S0= adr0
 │ n1 │
 ├────┤
 │ n2 │
 └────┘ <- SP@ = adr0-4
```

Ainsi, pour n valeurs, SPà délivre l'adresse adr0-(2*n):



La seule adresse fixe, dans le système FORTH, est donc S0. Une solution commode serait de s'en servir comme pointeur initial pour situer une valeur dans la pile. Exemple:

```
S0 4 - @
```

dépose la valeur n2 au sommet de la pile de données.

DEFINITION DE VARIABLES LOCALES

L'idée de départ est de reprendre le mécanisme des variables locales telles qu'elles sont gérées en langage PASCAL. Par souci de simplification, seul le mécanisme de base est conservé. Le but final est de pouvoir disposer d'un nom pour accéder à une valeur située à une position quelconque de la pile de données.

Première solution:
En utilisant l'adresse délivrée par S0, on peut définir des mots qui déposeront sur la pile de données la valeur d'un élément de la pile de données. La position de cet élément est calculée en appliquant la formule:

```
S0 - offset
```

où 'offset' désigne la position de l'élément dont on recherche la valeur.

Une proposition de définition symbolique pour cette première solution sera de la forme:

```
: VARLOC
CREATE      ( n --- <mot>)
  n 2 * ,   ( n*2=offset)
DOES>       ( --- contenu de S0-offset)
  @ S0 SWAP - @ ;
```

```
et 1 VARLOC VAL1   2 VARLOC VAL2
   5 7 3 65 8
   VAL1 ( empile 5)
   VAL2 ( empile 7)
   VAL1 ( empile 5)
```

Mais cette première solution ne permet pas la "localisation" des variables VAL1 et VAL2. En effet, toute définition utilisant ces mêmes variables (VAL1 et VAL2) utilisera aussi les mêmes valeurs. Or, la "localisation" doit permettre à chaque définition d'utiliser ses valeurs propres, ceci à partir des mêmes noms de variables.

Seconde solution:
On peut transférer l'adresse de la position du premier élément appartenant au groupe des variables locales au sommet de la pile de retour. Exemple:

```
SP@ >R
```

ce qui permet d'utiliser le contenu du sommet de la pile de retour comme pointeur initial à la place du contenu de S0:

```
RP@ @ ( ou R@ )
offset -
```

Mais cette solution comporte un inconvénient majeur, car l'empilement d'une valeur quelconque au sommet de la pile de retour sans un dépilement au sein de la même définition perturbe le fonctionnement de l'interpréteur FORTH. En outre, cette solution devient incompatible avec le mécanisme des boucles DO...LOOP qui utilisent également la pile de retour.

Troisième solution:
On transfère le pointeur de pile initial dans une variable:

```
VARIABLE POINTEUR
SP@ POINTEUR !
```

L'accès à une donnée empilée par la suite prend la forme:

```
: VARLOC
CREATE ( n --- <mot>) 1- 2* ,
DOES> ( --- donnée)
  @ POINTEUR @ SWAP - @ ;
```

Cette solution est compatible avec les mécanismes internes de l'interpréteur FORTH et des boucles DO...LOOP, mais ne permet toujours pas la "localisation". En fait, il faut disposer de deux mots supplémentaires, le premier chargeant POINTEUR avec l'adresse du pointeur de pile courant et sauvegardant la précédente valeur de POINTEUR, le second réalisant le mécanisme inverse, c'est à dire restaurant la valeur de POINTEUR précédement sauvegardée par le premier mot.

DES VARIABLES LOCALES A LA PELLE

Les variables dites locales, telles qu'elles sont définies dans les blocs 300 et 301, fonctionnent selon la troisième solution, à la différence près que POINTEUR désigne un tableau à une dimension et non une variable. L'accès au pointeur courant est réalisé par:

```
POINTEUR @
```

mais l'empilage et le dépilage du pointeur courant sont réalisés par les mots >POINT et POINT>. La profondeur de la pseudo-pile ainsi gérée est controlée par la constante PROFONDEUR, définie dans le bloc 300, et sa valeur est de 40 unités. Chaque groupe de variables locales empile le nombre d'arguments et le pointeur de pile courant sur POINTEUR, ce qui autorise jusqu'à dix niveaux de récursivité. Pour accroitre le nombre de niveaux, il faut augmenter la valeur initiale de cette constante.

```
SCR: 300
( VARIABLES LOCALES EN FORTH)

 40 CONSTANT PROFONDEUR
( Cette valeur peut etre modifiee afin)
( d'accroitre les niveaux d'appels.   )

VARIABLE POINTEUR  PROFONDEUR ALLOT

: >POINT ( n ---)
POINTEUR DUP 2+ PROFONDEUR <CMOVE
POINTEUR ! ;

: POINT> ( --- n)
POINTEUR @
POINTEUR DUP 2+ SWAP PROFONDEUR CMOVE ;

: ARGUMENTS  ( n ---)
DUP >POINT  2* SP@ + >POINT ;

: P:
CREATE  ( n ---)
  1- 2* ,
DOES>    ( --- [pointeur+n])
  @ POINTEUR @ SWAP - @ ;
```

Le mot ARGUMENTS peut être utilisé en interprétation ou en compilation. Lors de son exécution, il empile sur POINTEUR le nombre d'arguments et le pointeur de pile courant augmenté de la valeur d'offset correspondant au double du nombre d'arguments.

Le mot RESULT conserve les n éléments situés à partir du sommet de la pile de données, puis ramène le pointeur de pile courant à la valeur désignée par celui contenu dans POINTEUR et détruit les valeurs désignant le nombre d'arguments et le pointeur de pile qui sont situées au sommet de la pseudo pile POINTEUR.

Les en-têtes de variables locales sont définis à l'aide du mot 'P:'. Ce mot est défini en fin de bloc 300. Une version plus rapide à l'exécution est définie en assembleur FORTH 6809 dans le bloc 72.

```
SCR: 72
( VARIABLES LOCALES EN FORTH  )
( Version ASSEMBLEUR FORTH 6809)

: P:
CREATE  ( n ---)     1- 2* ,
;CODE    ( --- [pointeur+n])
x pshs  2 ,x ldd    d pshu
POINTEUR ldd   0 ,u subd   d x tfr
0 ,x ldd   0 ,u std   x puls
HEX 0099 jmp DECIMAL END-CODE
```

Le mot TO permet d'affecter une valeur numérique à l'un des éléments de la pile, cet élément étant désigné par le nom de la variable locale correspondante. Exemple:

```
 4 P: STO1
 55 1 ARGUMENT  TO STO1 0 RESULT
```

affecte la valeur 55 au premier élément d'adresse 'POINTEUR @ 4 2* + ', POINTEUR contenant l'adresse SPà avant l'empilage de l'argument. Le fonctionnement du mot TO est similaire au mot LIT, c'est à dire qu'il incrémente le contenu du sommet de la pile de retour de deux unités afin d'éviter l'exécution du mot qui le suit.

```
SCR: 301
: LOCALS: ( n --- <mot1 mot2 .. motn>)
1+ 1  DO
  I P:
LOOP ;

VARIABLE ITEMS

: RESULT  ( n ---)
DUP ITEMS !
BEGIN  ?DUP
WHILE . SWAP >R 1-  REPEAT
BEGIN  POINTEUR @ SP@ = 0=
WHILE  DROP         REPEAT
ITEMS @
BEGIN  ?DUP
WHILE  R> SWAP 1-  REPEAT
POINT> DROP
POINT> DROP ;

: TO ( n ---)
R@ @ 2+ @
POINTEUR @ SWAP - !
R> 2+ >R ;
```

Les en-têtes de variables locales peuvent être définis en série à l'aide du mot LOCALS: dont voici un exemple d'application. Soit à calculer la surface d'un parrallélipipède rectangle dont les arêtes sont désignées par COTEA, COTEB et COTEC:

```
 3 LOCALS COTEA COTEB COTEC
 : SURFACE  ( coteA coteB cotec --- aire)
  3 ARGUMENTS
 COTEA COTEB * 2*
 COTEA COTEC * 2* +
 COTEB COTEC * 2* +
  1 RESULT ;
```

Dans cet exemple, on définit trois variables locales, COTEA, COTEB et COTEC. A partir de ce moment, il faut équilibrer le contenu de la pile de données avec le même nombre d'éléments que de variables locales utilisées dans une définition. Les éléments sont affectés aux variables correspondantes dans le même ordre que celui de la création des variables locales. Ainsi, si on dépose successivement les valeurs 3 5 et 8 sur la pile, la

valeur 3 est affectée à COTEA, 5 à COTEB et 8 à COTEC. L'initialisation du POINTEUR est réalisée par l'exécution de 3 ARGUMENTS (normal, sinon POINTEUR pointe sur n'importe quoi...).

Le nombre d'éléments résiduels laissés sur la pile est déclaré par n RESULT. Dans l'exemple ci-dessus, 1 RESULT laisse la valeur située au sommet de la pile de données. Si on avait voulu conserver les valeurs de COTEA, COTEB et COTEC, il aurait fallu taper:

4 RESULT à la place de 1 RESULT

INTERET DES VARIABLES LOCALES

Les variables locales pourront avantageusement être utilisées dans les définitions gérant de nombreux paramètres. Bien que le temps d'exécution de la définition soit un peu plus long, les variables locales permettent une plus grande lisibilité des définitions y faisant appel, une grande facilité de maintenance et la gestion d'un plus grand nombre de paramètres au sein d'une même définition.

C'est cet aspect que je vais m'efforcer d'illustrer ci-après, en proposant un programme écrit en assembleur FORTH et en FORTH, tournant sur les systèmes THOMSON TO7 et TO7-70. Dans les routines qui suivent, on se propose de redéfinir les fonctions graphiques de dessin d'un point et d'un segment. Bien que ces routines soient déjà disponibles dans le moniteur et dans la cartouche FORTH (mots PSET et LINE), on y apportera les modifications suivantes:

- relogeabilité en mémoire du tracé en cours. En clair, possibilité de tracer des points ou des segments de droite ailleurs que dans la mémoire vidéo. L'intérêt de cette option sera expliqué plus loin.
- possibilité de définir des tracés en mode OR ou XOR. Le premier mode est celui normallement défini dans le moniteur THOMSON. Le second mode permet d'effacer un motif après tracé simplement en le retraçant une seconde fois.

Le programme commence dans le bloc 302. Les deux variables PLOTX et PLOTY permettent de mémoriser les coordonnées terminales d'un point tracé à l'écran.

Le mot PARAM est chargé d'affecter les valeurs xd et yd à PLOTX et PLOTY. Ce mot est un mot "mixte", en ce sens qu'il s'agit d'une primitive définie en code machine et est exécutable aussi bien sous FORTH qu'en tant que sous-programme dans une autre primitive définie également en code machine.

La variable STAD (pour STArt ADress) contient l'adresse du premier octet de la mémoire de l'écran vidéo. Toute modification de son contenu provoque le tracé du point ou du segment à une adresse différente. Si on tape 'HERE 200 + STAD !', le tracé du segment se fera dans une zone mémoire située 200 octets après le dernier mot du dictionnaire. Bien entendu, ce segment ne sera pas visible. Pour le voir, il faut faire ensuite un transfert de bloc:

```
COLOROFF
HERE 200 + DUP STAD !
8000 0 FILL
20 50 100 150 LINE
STAD @ 16384 8000 CMOVE
```

En implantant différents dessins à des emplacements variés (dans les banques mémoires par exemple), et en exécutant ensuite des transferts de blocs en série, vous pourrez réaliser une véritable animation graphique en temps réel (tellement temps réel qu'il faut même temporiser un peu entre deux transferts de blocs).

Le mot CALCUL délivre l'adresse de l'octet contenant le bit à modifier pour un tracé et dont les coordonnées ont été déposées au sommet de la pile de données.

```
SCR: 302

HEX 0099 CONSTANT NEXT
VARIABLE PLOTX   VARIABLE PLOTY

CODE PARAM  ( xd yd --- xd yd)
0 ,u ldd    PLOTY std
2 ,u ldd    PLOTX std    rts
FTH-RESOLVE    NEXT jmp  END-CODE

VARIABLE STAD  4000 STAD !

CODE CALCUL ( x y --- adr)
x y pshs   ,u++ ldy
y d tfr   28 # lda    mul
STAD addd             ( D=Y*40+STAD )
0 ,u ldx    x d exg
lsra rorb  ( lsrd       car x>255,donc  )
lsrb    lsrb        ( decalage 16 bits)
d,x leax   0 ,u stx    x y puls   rts
FTH-RESOLVE    NEXT jmp    END-CODE
DECIMAL
```

Dans le bloc 303, le mot TABIT contient des valeurs correspondant aux exposants de 2, ceci de 2E0 à 2E7. Le contenu de ce tableau est utilisé en suite sous assembleur.

La variable OPTION permet de choisir entre le tracé en mode OR ou XOR. Les options à affecter à OPTION sont données en commentaire dans le bloc correspondant. Exemple:

```
16384 STAD !  COLOROFF
1 OPTION !
50 100 120 150 LINE
50 100 120 150 LINE
```

trace et efface le segment. Si ce segment croise un motif déjà tracé, il efface les points précédemments allumés dans ce motif, puis lors du retraçage du segment, les points effacés sont restaurés. Visuellement, l'effet est très intéressant; prenons le cas d'un segment traversant un carré plein. l'intersection du segment efface les points communs au segment et au carré. Le retraçage du segment restaure le carré dans son état initial.

Le mot PLOTXY affiche un point aux coordonnées graphiques xd et yd, ceci en fonction du contenu de OPTION et dans une zone mémoire dont l'adresse de début est définie par le paramètre contenu dans STAD. Remarquez l'utilisation de la structure de contrôle if..then en assembleur FORTH, évitant l'emploi de labels pour les branchements.

```
SCR: 303
CREATE TABIT   HEX
80 C,  40 C,  20 C,  10 C,
 8 C,   4 C,   2 C,   1 C,

VARIABLE OPTION
( 0=> trace en OU INClusif logique  )
( 1=> trace en OU EXclusif logique  )

CODE PLOTXY  ( xd yd ---)
x y pshs
' PARAM jsr     ( PLOTX:=x,PLOTY:=y)
' CALCUL jsr    ( calcul adr(PLOTXY)
,u++ ldx   TABIT # ldy
PLOTX 1+ lda    7 # anda
a,y ldb   OPTION 1+ lda
1 # cmpa
eq if       0 ,x orb
   else     0 ,x eorb     then
0 ,x stb
x y puls   rts
FTH-RESOLVE   NEXT jmp   END-CODE
DECIMAL
```

Et pour finir, on définit pas moins de onze variables locales, XD, YD, XF, YF, DX, DY, XINCR, YINCR, CUMUL, X et Y. Celles-ci sont utilisées dans la définition de LINE dont l'exécution trace un segment de droite dont les coordonnées d'origine sont xd et yd et celles de fin xf et yf.

```
11 LOCALS:
XD YD XF YF DX DY XINCR YINCR CUMUL X Y
```

L'algorithme de tracé d'un segment de droite est celui dit 'méthode de LUCAS' et provient du livre 'Synthèse d'image: Algorithmes élémentaires' par G.HERON publié aux éditions DUNOD:

```
Génération SEGMENT-1 (xd,yd,xf,yf : entier)
Début co méthode de LUCAS fco
    entier : ΔX,ΔY, XINCR,YINCR,CUMUL,X,Y;
    x := xd, y := yd;
    afficher point (x,y);
    XINCR := si xd < xf alors +1 sinon -1;
    YINCR := si yd < yf alors +1 sinon -1;
    ΔX    := abs(xd-xf), ΔY := abs(yd-yf);
     si ΔX > ΔY co on dessine le maximum de points fco
       alors CUMUL := ΔX/2;
             Pour I := 1 jusqu'à ΔX faire
                 X := X + XINCR;
                 CUMUL := CUMUL + ΔY;
                 si CUMUL ≥ ΔX
                    alors CUMUL := CUMUL - ΔX;
                          Y := Y + YINCR;
                 fsi
                 afficher point (x,y);
             finpour
       sinon CUMUL := ΔY/2;
             Pour I := 1 jusqu'à ΔY faire
                 Y := Y + YINCR;
                 CUMUL := CUMUL + ΔX;
                 Si CUMUL ≥ ΔY
                    alors CUMUL := CUMUL - ΔY;
                          X := X + XINCR;
                 fsi
                 afficher point (x,y);
             finpour
     fsi
Fin
```

Cet algorithme, décrit dans ce livre, en page 22, est écrit en pseudo code (les PASCALiens n'auront aucune difficulté à traduire...) et donne en FORTH:

```
SCR: 304
: LINE ( xd yd xf yf ---)
0 0 0 0 0 0 0  11 ARGUMENTS
XD TO X    YD TO Y
X Y PLOTXY
XD XF <
IF  1 TO XINCR ELSE  -1 TO XINCR THEN
YD YF <
IF  1 TO YINCR ELSE  -1 TO YINCR THEN
XD XF - ABS TO DX
YD YF - ABS TO DY    DX DY >
IF   DX 2/ TO CUMUL    DX 1+ 1
     DO  X XINCR + TO X
         CUMUL DY + TO CUMUL
         CUMUL DX > CUMUL DX = OR
         IF   CUMUL DX - TO CUMUL
              Y YINCR + TO Y           THEN
     X Y PLOTXY                        LOOP
ELSE DY 2/ TO CUMUL    DY 1+ 1
     DO  Y YINCR + TO Y
         DX CUMUL + TO CUMUL
         CUMUL DY > CUMUL DY = OR
         IF   CUMUL DY - TO CUMUL
              X XINCR + TO X           THEN
     X Y PLOTXY                        LOOP
THEN 0 RESULT ;
```

La similitude d'écriture est flagrante. Si l'on n'avait pas fait appel aux routines de gestion de variables locales, mais à une gestion classique des paramètres de la pile, le mot LINE serait certainement illisible, car truffé de DROP, ROT, SWAP, PICK et autres cauchemars pour débutants et avec lesquels les rédacteurs d'articles peu scupuleux font trois chapitres d'initiation dans les revues d'informatique ayant la prétention d'apprendre quelque chose à leurs lecteurs. En aparté, cette démarche est aussi ridicule que de prétendre apprendre l'électronique en s'arrêtant au mode d'emploi d'un téléviseur.

La preuve est donc faite que des définitions FORTH peuvent être fort complexes et rester très lisible. Par cet exemple, on passe outre les problèmes de manipulation de la pile de données en créant des outils de gestion plus commodes.

ADAPTABILITE A D'AUTRES SYSTEMES

Vous pouvez adapter sans difficulté majeure les routines de gestion de variables locales à votre système. Voici les modifications à apporter:

- en FIG, remplacer CREATE par <BUILDS.

- si vous ne disposez pas de <CMOVE, voici comment définir ce mot:

```
: <CMOVE ( adr1 adr2 n --)
  ROT OVER PAD SWAP  CMOVE
        PAD ROT ROT  CMOVE  ;
```

Cette définition est donnée à titre de dépannage.

Le mot S0 n'est pas défini sur certains systèmes. Sa définition est alors la suivante pour les systèmes FIG (processeur Z80, 8080 et 6502):

```
6 USER S0
```

En ce qui concerne la réécriture du mot LINE en assembleur pour les systèmes équipés du 6809, elle est aisée à mettre en oeuvre. En effet, l'adressage indexé permet une méthode d'accès identique au mécanisme des variables locales. Dans ce cas, on peut se passer des noms de variables locales. Ceci peut être un bon exercice pour ceux qui nous ont commandé la disquette contenant l'assembleur FORTH (qui est, nous le rappelons, gratuite, excepté frais de disquette et d'envoi, soit 30,00 Fr au total, en timbres poste...).

Mais si vous avez un peu de patience, nous vous proposerons prochainement un nouveau langage, le BLAISE, qui est un véritable compilateur, issu de FORTH et dont la syntaxe est similaire au ... mais je vous laisse le soin de le découvrir par vous même. ■

Suite de la page 6

base de faits.

FUTURSYS introduit dans ce cas une relation d'ordre entre les faits qui seraient concurrents pour cette unification qui doit être unique, ce qui est possible, car l'utilisateur peut définir un ordre (transitif, comme pour les bases de structures) entre les faits qu'il définit ou que son application va définir.

COMBINATOIRE

Le mode combinatoire, au contraire, va effectuer toutes les unifications possibles de l'expressions. En l'absence de processeurs parllèles, chaque unification (et inférence) sera effectuée suivant l'ordre de la base de faits, quand le mécanisme de 'back-tracking' aura ramené l'unification au niveau de l'expression considérée. En effet, le mécanisme de 'back-tracking' permet de "défaire" les inférences, de la dernière qui a pu être effectuée à celle au niveau de laquelle il existe encore au moins une possibilité d'unification. ■

```
(**********************************************************)
(***  toolsl_2.pas    MAI 1986  (C)  LANGLOIS MICHEL  *****)
(**********************************************************)

procedure KeyDef(t,r,n,s,c : byte);
begin
   inline ($3A/T/               ( LD A,(T)                  )
           $F5/                 ( PUSH AF                   )
           $3A/N/               ( LD A,(N)                  )
           $47/                 ( LD B,A                    )
           $F1/                 ( POP AF                    )
           $F5/                 ( PUSH AF                   )
           $CD/$BE9B/           ( CALL #BE9B 'FIRMWARE'     )
           $BB27/               ( DEFW #BB27                )
           $3A/S/               ( LD A,(S)                  )
           $47/                 ( LD B,A                    )
           $F1/                 ( POP AF                    )
           $F5/                 ( PUSH AF                   )
           $CD/$BE9B/           ( CALL #BE9B 'FIRMWARE'     )
           $BB2D/               ( DEFW #BB2D                )
           $3A/C/               ( LD A,(C)                  )
           $47/                 ( LD B,A                    )
           $F1/                 ( POP AF                    )
           $F5/                 ( PUSH AF                   )
           $CD/$BE9B/           ( CALL #BE9B 'FIRMWARE'     )
           $BB33/               ( DEFW #BB33                )
           $06/$FF/             ( LD B,#FF                  )
           $3A/R/               ( LD A,(R)                  )
           $FE/$00/             ( CP #00                    )
           $20/$02/             ( JR NZ,+2                  )
           $06/$00/             ( LD B,#00                  )
           $F1/                 ( POP AF                    )
           $CD/$BE9B/           ( CALL #BE9B 'FIRMWARE'     )
           $BB39)               ( DEFW #BB39                )
end;

function inkey(t : byte): integer;
var n : integer;
begin
   begin
   inline ($3A/T/               ( LD A,(T)                  )
           $CD/$BE9B/           ( CALL #BE9B 'FIRMWARE'     )
           $BB1E/               ( DEFW #BB1E                )
           $3E/$FF/             ( LD A,-1       'A=0'       )
           $28/$01/             ( JR Z,+1                   )
           $79/                 ( LD A,C        'A=C'       )
           $32/N/               ( LD (N+1),A                )
           $AF/                 ( XOR A                     )
           $32/N+1)             ( LD (N),A                  )
   end;
   inkey:=N;
end;

procedure poke(ad : integer; vl : byte);
begin
   mem[ad]:=vl;
end;

procedure dpoke(ad,vl : integer);
begin
   mem[ad]  :=vl mod 256;
   mem[ad+1]:=vl div 256;
end;

function peek(ad :integer): byte;
begin
   peek:=mem[ad];
end;

function dpeek(ad :integer): integer;
begin
   dpeek:=peek(ad)+256*peek(ad+1);

end;

procedure DI;
begin
   inline   ($CD/$BE9B/         ( CALL $BE9B 'FIRMWARE'  )
             $BD04)             ( DEFW $BD04  DI         )
end;

procedure EI;
begin
   inline   ($CD/$BE9B/         ( CALL $BE9B 'FIRMWARE'  )
             $BD07)             ( DEFW $BD07  EI         )
end;
```

Voici une nouvelle série de "TOOLS" pour le système AMSTRAD/SCHNEIDER. Ils complètent ceux précédemment diffusés dans JEDI n°21.

Les TOOLS 1_2, 2_2, 3_2 sont à ajouter aux précédents. Dans TOOLS 1_2, les PEEK et POKE demandent une adresse en INTEGER, donc, si adr est supérieure à 32767, elle devient négative. Dans ce cas, il faut faire:

(65536-adresse)

Le TOOLS 6 est un DUMP pour imprimante MANESSMAN MT80.

Le TOOLS7 donne la fonction TIME en secondes. Lors de l'écriture de WRITE(TIME), ce sera dans le format $0.0E^{n}$, et je ne comprends pas pourquoi...

```
function joy0 : byte;
var n : integer;
begin
   begin
   inline (#CD/#BE9B/                    ( CALL #BE9B 'FIRMWARE' )
           #BB24/                        ( DEFW #BB24            )
           #32/N)                        ( LD (N),A              )
   end;
   joy0:=n;
end;

function joy1 : byte;
var n : integer;
begin
   begin
   inline (#CD/#BE9B/                    ( CALL #BE9B 'FIRMWARE' )
           #BB24/                        ( DEFW #BB24            )
           #7D/                          ( LD A,L                )
           #32/N)                        ( LD (N),A              )
   end;
   joy1:=n;
end;

(*************************************************************)
(****  tools2_2.pas    MAI 1986  (C)   LANGLOIS MICHEL   *****)
(*************************************************************)


function xpos: integer;
var n : integer;
begin
   begin
   inline (#CD/#BE9B/                 ( CALL #BE9B 'FIRMWARE'    )
           #BBC6/                     ( DEFW #BBC6               )
           #ED/#53/N)                 ( LD (N),DE                )
   end;
   xpos:=n;
end;

function ypos: integer;
var n : integer;
begin
   begin
   inline (#CD/#BE9B/                 ( CALL #BE9B 'FIRMWARE'    )
           #BBC6/                     ( DEFW #BBC6               )
           #22/N)                     ( LD (N),HL                )
   end;
   ypos:=n;
end;

(*************************************************************)
(****    tools3_2.pas   MAI 1986  (C)    LANGLOIS  MICHEL ****)
(*************************************************************)

function hpos : integer;
var n : integer;
begin
   begin
   inline (#CD/#BE9B/              ( CALL #BE9B 'FIRMWARE'      )
           #BB78/                  ( DEFW #BB78                 )
           #7D/                    ( LD A,L                     )
           #32/N)                  ( LD (N),A                   )
   end;
   hpos:=n;
end;

function vpos : integer;
var n : integer;
begin
   begin
   inline (#CD/#BE9B/              ( CALL #BE9B 'FIRMWARE'      )
           #BB78/                  ( DEFW #BB78                 )
           #7C/                    ( LD A,H                     )
           #32/N)                  ( LD (N),A                   )
   end;
   vpos:=n;
end;

function readchar(x,y :byte): char;
var n : char;
begin
   gotoxy(x,y);
   begin
   inline (#CD/#BE9B/              ( CALL #BE9B 'FIRMWARE'      )
           #BB60/                  ( DEFW #BB60                 )
           #32/N)                  ( LD (N),A                   )
   end;
   readchar:=n;
end;
```

```
(************************************************************)
(***  tools6.pas     MAI 1986   (C)   LANGLOIS MICHEL    ******)
(************************************************************)


procedure dump(r,p :integer);  { dump(0,p) normal, (1,p) inverse }
                               { p equal couleur papier         }
var x,y,k: integer;
    flag,bit: byte;
begin
 writeln(lst,#27'A'+CHR(6));
 x:=399;
 while x>7 do
   begin
    write(lst,#27'K'+CHR(127)+CHR(2));
     for y:=0 to 638 do
      begin
      bit:=0; flag:=64;
       for k:=0 to 6 do
        begin
         gmove(y,x-k);
          if testr(0,0)<>p then bit:=bit or flag;
          flag:=flag shr 1;
        end;
       if r=1 then bit:=127-bit;
       write(lst,chr(bit));
      end;
    write(lst,^J^M);
    x:=x-7;
   end;
end;

(************************************************************)
(***  tools7.pas     MAI 1986   (C)   LANGLOIS MICHEL      ****)
(************************************************************)


procedure setTime(t :real);    { t en secondes                  }
var de,hl : integer;
begin
 t:=t*300;
 de:=trunc(t/65536.0);
 hl:=round(t-65536.0*de);
begin
   inline (#ED/#5B/de/         { LD DE,(DE)                     }
           #2A/hl/             { LD HL,(HL)                     }
           #CD/#BE9B/          { CALL #BE9B  'FIRMWARE'         }
           #BD10)              { DEFW #BD10                     }
   end;
end;

function time :real;           { donne le temps en secondes     }
var de,hl   : integer;         { ecrire write(time 0:0)         }
    dde,hhl : real;            { sinon c'est ecrit 0.0E1        }

begin
   begin
        inline (#CD/#BE9B/     { CALL #BE9B  'FIRMWARE'         }
                #BD0D/         { DEFW #BD0D                     }
                #ED/#53/DE/    { LD (DE),DE                     }
                #22/HL)        { LD (HL),HL                     }
   end;

if de<0 then dde:=(de+65536.0)*65536.0
        else dde:=de*65536.0;

if hl<0 then hhl:=(hl+65536.0)
        else hhl:=hl;

time:=(dde+hhl)/300.0;

end;
```

INTRODUCTION

Cet article est paru dans le magazine MICRO REVUE no 10, janvier-février 1986, et sa reprise dans nos colonnes est autorisée par l'auteur et le directeur de MICRO-REVUE. Nous allons étudier comment définir un langage et réaliser un compilateur pour ce langage.

1er EPISODE: LA DESCRIPTION SYNTAGMATIQUE D'UN LANGAGE

Nous allons nous intéresser, pour l'instant, uniquement à la description de la syntaxe d'un langage: le niveau sémantique n'intervient pratiquement que dans l'étape finale, la réalisation du compilateur.

Pour décrire un langage, nous allons avoir besoin d'un autre langage, un méta-langage. Nous utiliserons ici la notation BNF (Backus Naur Form), valable principalement pour les langages 'classiques' et structurés, donc inappropriée pour le FORTH ou le BASIC. Sa première utilisation fut pour définir l'ALGOL 60; FORTRAN, par exemple, avait les caractéristiques de son compilateur et non l'inverse.

Un langage se définit alors par L(T,N,P,S), T étant l'ensemble des éléments terminaux (par exemple GOTO), N l'ensemble des éléments non terminaux (les catégories grammaticales), P l'ensemble des productions (règles de grammaire) et S un élément de N, l'élément de départ.

On utilise pour décrire P les métasymboles suivants:

- ::= pour une production: $A::=\gamma$ signifie "réécrire γ pour A".
- | pour le "ou"
- <> pour symboliser un non terminal
- {--} pour une répétition incluant la séquence vide ε .

Prenons un exemple: soit le langage

<phrase> ::=<sujet><verbe>
<sujet> ::= le chat | le chien
<verbe> ::= mange | dort

Les quatres phrases syntaxiquement correctes sont donc:

le chat mange
le chien mange
le chat dort
le chien dort

On a S=<phrase>, T={le chien, le chat, mange, dort} et N={<phrase>,<sujet>,<verbe>}.

Dans la suite, les majuscules désigneront les non terminaux, les minuscules les terminaux et les caractères grecs des séquences d'éléments de TUN (des éléments de TUN*).

On dit qu'une séquence η peut etre directement générée à partir d'une séquence γ s'il existe des séquences α, β , γ' , η' $\gamma'::\eta'$. On note alors $\gamma \rightarrow \eta$.

Une séquence σ_n peut alors etre générée à partir d'une séquence σ_0 s'il existe $\sigma_1..\sigma_{n-1}$ telles que $\forall i \in [1..n], \sigma_{i-1} \rightarrow \sigma_i$. On note $\sigma_0 \overset{*}{\rightarrow} \sigma_n$ des séquences de terminaux générables à partir de S:

$$L=\{\gamma / S \overset{*}{\rightarrow} \gamma \text{ et } \gamma \in T^*\}.$$

L'étape finale de la génération ne contient plus que des terminaux, d'où leur nom.

On dit qu'un langage est sensible au contexte si la substitution d'un élément non terminal tient compte des autres symboles présents, par exemple $\alpha A \beta ::= \alpha\gamma\beta$.

Au contraire, si les productions sont de la forme $A::=\gamma$, il est dit hors contexte, ou non sensible au contexte. Il est très simple d'analyser les phrases d'un tel langage, pour vérifier si elles appartiennent effectivement au langage. On utilise l'analyse descendante, avec lecture d'un symbole à la fois (LL1) : on reconstruit la phrase à partir du symbole de départ. Par exemple:

<phrase>	le chien mange
<sujet> <verbe>	le chien mange
le chien <verbe>	le chien mange
<verbe>	. mange
mange	mange
-	-

Mais il faut prendre quelques précautions si l'on ne veut pas que l'analyse échoue par erreur d'aiguillage et éviter les retours en arrière: considérons le langage

S::=A|B
A::=xA|y
B::=xB|z

analysons xz:

S	xz
A	xz
xA	xz
A	z

et notre algorithme se plante. Le problème réside dans la première étape où il est impossible de décider entre A et B puisqu'on ne lit qu'un symbole à la fois. Une solution serait de remonter et de prendre B, mais on a dit qu'on ne voulait pas de retour en arrière. Il va donc falloir prendre des précautions dans les définitions des langages; énonçons la loi:

Si A:: $\gamma_1 | \gamma_2 | ... | \gamma_n$
on doit avoir $i \neq j \Rightarrow first(\gamma_i) \cap first(\gamma_j) = \emptyset$

$first(\gamma)$ est l'ensemble des symboles terminaux qui peuvent apparaitre en première position dans les phrases dérivées de γ .

Dans l'exemple précédent, redéfinir le langage par .

S::=C | xS
C:: y | z

élimine tout problème

Le langage suivant pose un autre problème:

S:: Ax
A:: x | ε où ε représente la cahine vide.

Si on analyse x, on obtient:

S	x
Ax	x
xx	x
x	

C'est le problème de la chaine nulle. Pour l'éviter, énonçons:

Si $A \overset{*}{\longrightarrow} \varepsilon$
il faut $first(A) \cap follow(A) = \emptyset$

avec, en considérant toutes les productions Pi: Xi:: $\gamma_i A \eta_i$ $folow(A) = \bigcup_i first(\eta_i)$.

Ainsi, pour générer B,BB,BBB,.....

A:: B | AB viole la loi 1
A:: ε | AB viole la loi 2

On voit donc qu'il est impossible de faire des définitions récursives à gauche, seule la récursivité à droite est possible, c'est à dire A:: ε | BA, qui équivaut d'ailleurs à A:: {B}.

→

Ces deux lois sont suffisantes pour éviter tout prblème au cours de l'analyse.

Remarquons que, tout à l'heure, nous avons remodelé un langage pour lui faire respecter nos deux lois. Mais attention, considérons l'exemple suivant:

```
S::=A | S-A
A::=a | b | c
S'::=A | A-S'
```

On a S—*—> a-b-c et S'—*—> a-b-c, mais le premier corrspond en fait à (a-b)-c et le second à q-(b-c), ce qui est différent. Quand on définir un langage par sa structure syntaxique, il faut quand même penser à sa structure sémantique; la première réflète la seconde.

Vous voyez maintenant pourquoi je vous disais que cette notation ne s'appliquait pas au FORTH. Cela donnerait quelque chose du genre <mot>::= : {<mot>};.

Le mois prochain, je vous parlerai des diagrammes syntaxiques, une autre façon de décrire un langage.





En caricaturant un peu, on pourrait dire qu'il existe deux grandes variétés de systèmes informatiques : ceux qui vous demandent systématiquement l'heure sans être sûrs d'en avoir besoin, et ceux qui vous la donnent systématiquement au cas où vous pourriez en avoir besoin.

Bien sûr, nous renverrons dos à dos ces deux conceptions, car le problème sous-jacent ne se réduit pas à une simple question anecdotique d'ergonomie. Il s'agit d'une véritable "attitude" du logiciel tant vis à vis de la machine-hôte que de l'homme utilisateur, caractérisée par davantage d'autonomie, d'aménité, voire de ... prévenance.

C'est bien la pression des utilisateurs, las de se heurter sans cesse aux mêmes difficultés, ou de devoir répondre à chaque utilisation aux mêmes sempiternelles questions, qui pousse les réalisateurs de logiciels à rendre ceux-ci plus conviviaux et dans les meileurs cas à leur donner une apparence d'intelligence.

Le langage L.P.B. n'est donc ni le premier, ni le seul à prendre en compte de façon systématique ces différentes préoccupations, mais il cherche à prendre en compte le mieux possible trois mécanismes fondamentaux que nous allons tenter d'expliciter.

1/ LE PREMIER MECANISME :

```
---------------------------------------------
' Le Logiciel doit reconnaître lui-même    '
' son environnement, afin de de pouvoir    '
' lui adapter son comportement.            '
---------------------------------------------
```

Un logiciel bien conçu devrait être capable de répondre de lui-même aux questions suivantes :

- sur quel type de machine suis-je ?
- combien de mémoire vive est à ma disposition ?
- quel genre d'écran ?
- quel genre de clavier ?
- quelle imprimante est connectée ?
- quelle heure est-il ?
- où sont mes fichiers auxiliaires ?

Bien que les algorithmes à mettre en oeuvre soient souvent relativement simples, on voit encore trop de logiciels, même de prix, incapables de se débrouiller seuls devant des questions ausi simples. Qu'ils en soient "conscients" au point de demander secours à l'opérateur est un pis-aller acceptable. Dans d'autres cas, on exigera qu'un spécialiste vienne préalablement manipuler de mystérieux modules appelés INSTALL ou CONFIG, dont l'utilité est de décrire au départ, à l'intention du logiciel principal, le type de configuration sur lequel on va lui demander de fonctionner.

La pire situation est qu'un logiciel L fonctionnant convenablement sur une première machine M1 se trouve incapable de fonctionner sur une autre machine M2, semblable ou réputée compatible, et ce, sans qu'un utilisateur raisonnablement compétent en comprenne la raison.

La situation idéale serait que le logiciel dispose parmi ses données d'une decription des principaux environnements au sein desquels on est susceptible de lui demander de fonctionner, lui indiquant les variantes et les sous-variantes à traiter. Une telle information est souvent directement ou indirectement accessible. Par exemple, la plupart des interpréteurs BASIC écrits par MICROSOFT commencent par déterminer l'étendue de la mémoire RAM diponible en essayant successivement de modifier et de relire un à un le contenu de tous les octets théoriquement adressables. Lorsque ce test échoue

sur un octet, il en conclut que la fin de mémoire est rencontrée et fournit à l'utilisateur un message du type :

MICROSOFT BASIC Version n.n   xxxxx BYTES FREE.

Cette technique est bien autre chose qu'un gadget : c'est aussi un moyen de vérifier le bon fonctionnement de la mémoire.

De tels auto-diagnostics contribuent à la fiabilité des systèmes, mieux que le recours à un opérateur humain. Le logiciel de navigation d'un boeing de la KOREAN AIR LINES est capable de reconstituer en toute sécurité et à tout instant, grâce aux données recoupées de trois centrales inertielles, la position de l'avion, à condition que la position initiale lui ait été communiquée. Si le logiciel avait été capable de se procurer par lui-même les coordonnées de l'aéroport de départ sans les demander à l'équipage, il n'y aurait pas eu de risque qu'une erreur de saisie ne le conduise par inadvertance à se promener au-dessus des îles Sakhaline.

L'intelligence artificielle est à la mode. Nous invitons nos lecteurs, informaticiens avertis, à s'assurer que les logiciels ambitieux qu'ils manipulent commencent bien par contenir ce petit système expert qui va leur permettre de se situer eux-mêmes dans l'ensemble des configurations auxquels on les destine.

Plus le langage dans lequel est écrit le logiciel est lui-même portable, plus cette question est importante, car avec le développement des banques de logiciels et du téléchargement, un logiciel a besoin de savoir sur quel genre de machine il a "atterri". Pour être concret, nous donnons ci-joint le canevas d'un heuristique permettant de reconnaître des machines aussi différentes que MSX, IBM-PC, TO7, MO5 et AMSTRAD.

```
10 GOSUB 1000
20 PRINT "La machine-support est un.............  ";MA$
30 PRINT "Modèle................................  ";MO$
40 PRINT "Micro-processeur .....................  ";MI$
50 END
1000 REM ----- heuristique de reconnaissance de la machine support
1001 REM ------( à perfectionner et compléter )------------------
1002 REM
1010 A=PEEK(65532):MA$="Commodore":MI$="6502"
1020 IF A=22  THEN MO$="CBM 8000":RETURN
1030 IF A=226 THEN MO$="C64":RETURN
1040 IF A=34  THEN MO$="VIC 20":RETURN
1050 IF PEEK(1)=195 THEN MA$="MSX":MI$="Z80":RETURN
1060 A=PEEK(&HFFFE):MA$="THOMSON":MI$="6809"
1070 IF A=&HF9 THEN MO$="TO7":RETURN
1080 IF A=&HFO THEN MO$="MO5":RETURN
1090 IF PEEK(0)=0 THEN MA$="IBM":MO$="PC-XT":MI$="80-88":RETURN
1100 A=PEEK(&HC002):MA$="AMSTRAD":MI$="Z80"
1110 IF A=0 THEN MO$="CPC 64":RETURN
1120 IF A=1 THEN MO$="CPC 128":RETURN
1130 IF A=2 THEN MO$="CPC 6128":RETURN
1140 IF A=3 THEN MO$="CPC 6256":RETURN
1150 MA$="Inconnu":MO$="Inconnu":MI$="Inconnu"
1199 RETURN
```

2/ LE DEUXIEME MECANISME :

```
 -------------------------------
'  Le Logiciel doit être capable '
'  de migrer de lui-même vers    '
'  un autre environnement.       '
 -------------------------------
```

A quoi bon reconnaître son environnement, diront certains, alors qu'il y a toujours un opérateur qui est là pour le lui dire ? Et à quoi bon laisser à un logiciel l'intiative de migrer tout seul, alors qu'il est si facile d'utiliser la commande COPY, qui universellement disponible sous tous les systèmes d'exploitation ? Enfin n'est-il pas dangereux de laisser une telle liberté à un logiciel ?

Ces questions sont importantes, et sont encore au coeur du débat pour la conquête de l'espace, par exemple, où tenants et adversaires des vaisseaux habités s'affrontent.

Dans des cas plus simples, le bon sens a imposé la solution. Les opératrices du téléphone ont disparu. Si on laisse encore des humains dans les cabines de contrôle des rames du métropolitain, c'est surtout pour rassurer les passagers, et que peut faire un pilote de TGV lancé à vive allure qui aperçoit au dernier moment un obstacle sur la voie, sinon augmenter le nombre des victimes ?

Se copier sur un autre support suppose que notre logiciel a été capable de reconnaître à la fois l'environnement de départ (celui sur lequel il se trouve), et l'environnement de destination, par exemple : une disquette, un disque dur, une liaison RS232, une imprimante. Quoi de plus naturel enfin qu'un logiciel que vous venez d'acheter dans le commerce et dont vous lancez l'exécution sur votre machine se dise la première fois : " Chic, il y a un disque dur NEC de 20 Millions d'octets, je vais m'installer dessus et rejoindre mes copains qui y sont déjà, et dont je vais pouvoir partager les utilitaires". S'il est bien élevé, il le fera, et vous en informera.

En guise d'illustration, nous donnons une organisation-type permettant à n'importe quel programme écrit en BASIC :

- de s'exécuter, par l'instruction RUN (classique)
- de s'identifier, grâce à une ligne 2 (DATA)
- de se sauver sur disquette, par l'instruction GOSUB 3
- de s'expédier lui-même en format ASCII vers un support externe (GOSUB 4)
- de se faire lister sur une imprimante type EPSON (GOSUB 5)

```
1 GOTO 10 :REM Noyau de primitives de transfert
2 DATA VOYAGE
3 RESTORE:READ F$:SAVE F$:RETURN
4 RESTORE:READ F$:SAVE "SPOO:"+F$+".ASC",A:RETURN
5 LPRINT CHR$(27)"3"CHR$(16)CHR$(27)"SO"CHR$(15):LLIST:RETURN
10 .... (corps du programme) ....
```

3/ LE TROISIEME MECANISME

```
 ---------------------------------
 '  Le logiciel se reproduit, et  '
 '  en profite pour s'améliorer.  '
 ---------------------------------
```

La reproduction est un merveilleux mécanisme par lequel le monde animal et végétal se distingue du monde inanimé des minéraux.

On trouve certes dans le monde inanimé de veritables chefs d'oeuvres : les cristaux, les stalactites, les arcs-en-ciels, les aurores boréales.

Mais la vie, c'est autre chose, c'est un secret bien caché au sein des molécules d'acide désoxyribonucléique (ADN) que la science moderne perce petit à petit.

Une molécule vivante est une molécule capable de lire un code contenu en elle-même de façon à construire, à partir de matériaux initialement inertes, une autre molécule elle aussi douée de vie.

De façon analogue, on peut concevoir un logiciel ayant des propriétés telles qu'il serait capable de se lire lui-même de façon à créer à partir d'un zone de mémoire RAM au départ inerte, sur une machine éventuellement différente, un autre logiciel possédant les mêmes propriétés.

A ne pas confondre surtout avec un simple clonage, obtenu en informatique par la commande COPY, qui est une forme de reproduction particulière sans grand intérêt ici. La reproduction dont nous parlons maintenant est une véritable fécondation, où l'enfant n'est pas identique au parent, puisque les données innées lues dans le code génétique se mêlent à des données propres au terrain qui a servi de théâtre à l'opération. Le fruit de la fécondation n'est pas toujours viable. Il n'est pas exempt de défaut. Seuls les meilleurs survivent, et parfois, l'enfant surpasse les parents.

Et qui va imposer la sélection naturelle, indispensable instrument de l'amélioration de cette nouvelle race des logiciels ? ...C'est l'utilisateur humain bien sûr, paresseux par nature, qui favorisera le développement des logiciels les plus efficaces, c'est-à-dire ceux qui lui épargneront le mieux sa propre peine. Au début tout au moins, car on peut imaginer dans un second temps que les logiciels se combattent eux-mêmes entre eux pour occuper les meilleures places dans les espaces informatiques qui leur seront accessibles.

Le compilateur BALCOM, qui traite le langage L.P.B., obéit à cette logique, en ce sens qu'il intègre en son sein les trois mécanismes qui viennent d'être présentés. Véritable méta-compilateur, inspiré en droite ligne de théories dont les bases furent jetées il y a déjà vingt ans par MM. KNUTH et WIRTH, il est effectivement capable de générer pour une machine nouvelle une nouvelle version qui reproduira les mêmes caractéristiques génétiques. A ce titre, il est préférable de parler de la famille des compilateurs BALCOM. Cette famille ne demande qu'à s'agrandir, et dans trois directions :

- Sur une machine donnée, en traitant un langage-source de plus en plus évolué, tendant asymptotiquement vers le langage humain (c'est le projet ELENA, déjà opérationnel à partir d'une souche APL).

- Vers de nouvelles machines, équipées le cas échéant de nouveaux micro-processeurs, de nouvelles architectures, de nouveaux langages.

- Sur une machine donnée, et sur un même langage donné, en embarquant des fonctionnalités supplémentaires : il n'y a pas que la fonction de reproduction dans un être vivant.

Bien que les développements qui viennent d'être cités soient potentiellement explosifs, il faut bien reconnaître que jusqu'ici la diffusion de L.P.B. est restée relativement confidentielle, et que loin d'être absolument originales, les caractéristiques citées se retrouvent dans nombre d'autres langages, tels que le FORTH.

Vus sous cet angle, les langages évolués souffrent d'un handicap : on peut écrire un interpréteur BASIC en BASIC, un compilateur FORTRAN en FORTRAN, et de même pour COBOL, APL ou ADA, mais ils seront abominablement lents, et dans une compétition de type biologique, ils sont voués comme les monstres préhistoriques à l'extinction par sélection naturelle.

Parmi les classiques, seuls les langages assembleur et C peuvent être considérés comme capables de se recompiler eux-mêmes à 100% et donc de s'adapter à des variations rapides de l'environnement.

Par comparaison, un compilateur L.P.B. typique aujourd'hui occupe 8 kilo-octets, et est capable de se compiler lui-même, donc de se régénerer, en trente secondes sur un micro-ordinateur 8 bits. Les versions 16 et 32 bits, qui ne sont pas encore opérationnelles, auront une gestation un peu plus lente, mais une puissance nettement plus importante.

L'ancêtre de la famille n'occupait que 6K dans un antique TRS-80. Mais au fait ... d'où venait-il celui-là ? ■

L'A.P.L SUR MICRO-ORDINATEURS

On assiste actuellement à une migration du langage APL des gros ordinateurs vers les micros. En effet, les conditions nécessaires au support d'un langage APL sont maintenant réalisées sur micro, à savoir:

- une puissance de calcul suffisante avec les processeurs 8088+8087 ou 68000.

- une capacité mémoire suffisante. Il faut compter 128K RAM pour une configuration d'initiation, et 256K pour pouvoir travailler plus sérieusement.

Une condition pratique qui a également freiné le développement de l'APL est la nécessité de générer à l'écran un jeu de caractères spécifique et de configurer le clavier en conséquence. Le problème du clavier se résoud généralement facilement avec des autocollants. Celui de l'écran est plus délicat.

LANGAGES APL ACTUELLEMENT DISPONIBLES SUR MICROS

APL sur TRS-80

L'APL-80 de la société RAMWARRE sur TRS-80 a été un des premiers (vers 1982 ?) APL sur micro. Les caractères spécifiques à l'APL étaient remplacés par des lettres réservées du jeu de caractères standard, ce qui permettait d'éluder le problème. Aucune modification du clavier n'était nécessaire. L'Espace de Travail (zone mémoire disponible pour programmer) était e 26 K, ce qui peut suffire pour une initiation au langage.

APL sous CP/M

L'APL V-80 de VANGUARD pouvait, en principe, tourner sur tout micro supportant le CP/M. La résolution des problèmes liés au jeu de caractères APL était... laissée à la discrétion de l'utilisateur. Avec un espace de travail de 27K, ce progiciel n'a pas connu un grand succès.

Le VIZ-APL, d'origine anglaise, apparu il y a environ deux ans utilisait une pagination sur disque pour s'affranchir des limites d'adressage du Z80 et gérer un espace de travail plus conséquent. Ce langage, qui devait être distribué en France par la CISI a disparu de la circulation.

L'APL sous MS-DOS

Il y a quelques années apparurent presque simultanément l'APL*PLUS/PC de STSC et le PC APL d'IBM. Actuellement, le premier en est à sa version 5 et le second à sa version 2. L'APL*PLUS nécessite un changement de la PROM du générateur de caracères (fournie avec le logiciel). L'APL IBM génère les caractères par soft à condition de disposer d'une interface écran CGA ou EGA, ou l'équivalent pour les compatibles. Tous deux utilisent le co-processeur 8087 s'il est présent. La configuration mémoire minimum est de 256K et 512K sont conseillés, mais c'est là une configuration quasiment standard actuellement.

L'APL sous UNIX

Le DIALOG APL de Dyadic Systems (anglais) est en principe portable sur tout ordinateur sous UNIX System V. Réservé surtout aux applications multipostes, il n'est pas à la portée d'un particulier.

L'APL sur machine dédiée

La société SOFREMI importe l'AMPERE, un portatif japonais muni d'un écran à cristaux liquides, et doté de l'APL68000, un logiciel déja connu sur des machines à base de processeur 68000 (Thorn-EMI). La mauvaise lisibilité de l'écran et le fait que cette machine soit seulement dédiée à l'APL ont fait qu'elle n'a pas connu un grand succes.

L'APL sur MacINTOSH

En 1985 est apparu le PortaAPL de la société Portable Software, autorisant un espace de travail de 230Ko sur MacINTSOSH-512K. Les possibilités graphiques du Mac ont permis de résoudre au mieux le problème des caractères APL.
D'autre part, l'APL-90 (du SM-90 de Bull) a également été porté sur MacINTOSH.
Une adaptation de l'APL68000 a également été annoncée.

L'APL sur QL SINCLAIR

L'évennement le plus intéressant pour les APListes peu fortunés a été en 1985 l'apparition du QL/APL, une version de l'APL68000 portée sur QL par la société anglaise MicroAPL. Il y a deux versions. L'une utilise des mots clefs réservés pour remplacer les caractères spéciaux APL. L'autre génère ces caractères à l'aide d'une PROM.
Avec les 128K de base du QL on a un espace de travail de 28K, ce qui est peu, mais suffit pour une initiation au langage? Toutefois, la mémoire du QL peut être augmentée jusqu'à 640Ko, et l'espace de travail est alors supérieur à 500Ko, ce qui est considérable. Une configuration à 256ko parait raisonnable.

Vous pouvez maintenant pratiquer l'APL avec un investissement d'environ 6000 Fr (matériel + logiciel). L'ADULA (Association des Utilisateurs du Langage APL) organisait pour ses membres des achats groupés pour bénéficier des meilleurs prix sur le logiciel et le matériel. Malheureusement, l'avenir du QL est désormais incertain.

F. ESPINASSE

QUELQUES PRIX

Voici à titre indicatif quelques prix très approximatifs seulement destinés à permettre au lecteur de se faire une idée. Ces prix demandent à être confirmés auprès des distributeurs.

QL/SINCLAIR 128K RAM	4500 Fr
QL/APL	1500 Fr
PortaAPL (pour MacINTOSH)	3500 Fr
APL IBM-PC version 2	3500 Fr
APL*PLUS/PC version	6000 Fr

QUELQUES ADRESSES UTILES

UNIWARE	8, rue Boileau 75016 PARIS 45272061 et 45272071 pour APL*PLUS/PC et PortaAPL (Mr PERRAILLON)
G.T.I.	17, rue de la Croix-Nivert 75015 PARIS 42732323 pour APL*PLUS/PC et APL IBM-PC (Mr BRUILLARD)
CISI	35 Bd BRUNE 75014 PARIS 45458546 pour Dyalog APL (Mr Beyda)
SYNC	12, place de l'Hotel de Ville 42000 ST ETIENNE pour APL-90 71326562
LOGICIA	La Butte aux Creches, Chemin du Vallot 78350 JOUY EN JOSAS 39560574 pour QL/APL
ADULA	21, place de Bretten 91160 LONGJUMEAU 42066330 pour achat QL et QL/APL (Mr J.M. FALQ)